Algorithms and Data Structures 2
CS 1501

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Lab 9 and Homework 9: next Monday 11/21 @ 11:59 pm
  - Assignment 3: ~~Monday 11/28~~ Friday 12/9 @ 11:59 pm
  - Assignment 4: Friday 12/9 @ 11:59 pm

Recap …

- Greedy algorithms
  - elegant but hardly correct
  - optimal substructure
  - greedy choice property
- Without the greedy choice property
  - have to solve all subproblems
  - can be done recursively
- Memoization
  - still recursive
  - avoid solving the same subproblem twice

Recap …

- Dynamic Programming
  - avoid solving the same subproblem twice
  - iterative:
    - start with smaller subproblems then larger subproblems, …
  - sometimes possible to optimize space needed

Recap …

- Fibonaaci
  - inefficient recursive solution
  - memorization solution
  - dynamic programming
    - with space optimization

Solving Dynamic Programming Problems

- Can you solve the problem using subproblems?
  - What is the first decision to make to solve the problem?
  - What subproblem(s) emerge out of the that first decision?
- Can you make the first decision without having to wait for the solution of the subproblems?
  - If yes, that’s a greedy algorithm! Congratulations!

Solving Dynamic Programming Problems

- If you have to wait for subproblem solutions to make the first decision, try the following steps
- start with a recursive solution
- if inefficient, do you have overlapping subproblems?
- identify the unique subproblems
- solve them from smaller to larger
- This is dynamic programming!
- Optimize space if possible

This Lecture

- Dynamic Programming Problems
  - Unbounded Knapsack
  - 0/1 Knapsack
  - Subset Sum
  - Edit Distance
  - Longest Common Subsequence

The unbounded knapsack problem

Given a knapsack that can hold a weight limit L, and a set of n types items that each has a weight (w_i) and value (v_i), what is the maximum value we can fit in the knapsack if we assume we have unbounded copies of each item?

weight:	6	3	4	2
value:	30	14	16	9

10 lb. capacity

A greedy algorithm

- Try adding as many copies of highest value per pound item as possible:
  - Water: 30/6 = 5
  - Rope: 14/3 = 4.66
  - Flashlight: 16/4 = 4
  - Moonpie: 9/2 = 4.5
- Highest value per pound item? Water
  - Can fit 1 with 4 space left over
- Next highest value per pound item? Rope
  - Can fit 1 with 1 space left over
- No room for anything else
- Total value in the 10 lb knapsack?
  - 44
    - Bogus!

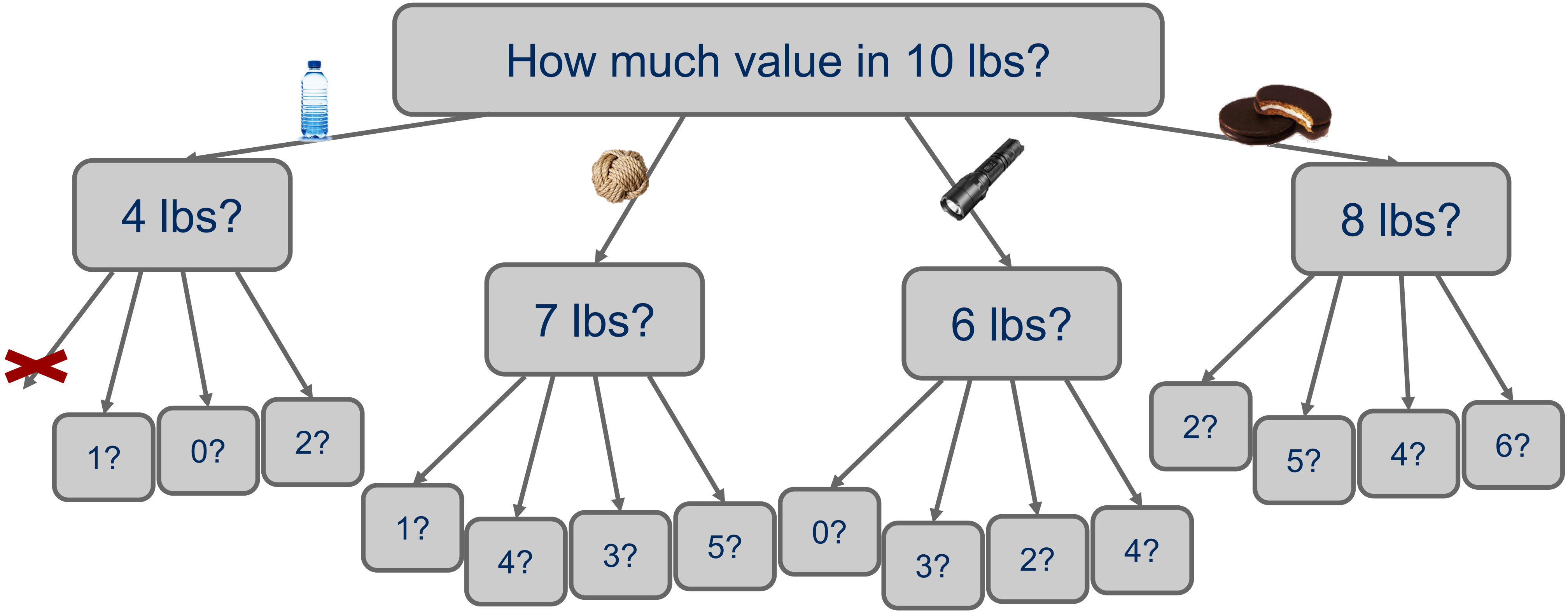
But why doesn’t the greedy algorithm work for this problem?

The greedy choice property is missing!

Recursive Solution

weight:	6	3	4	2
value:	30	14	16	9

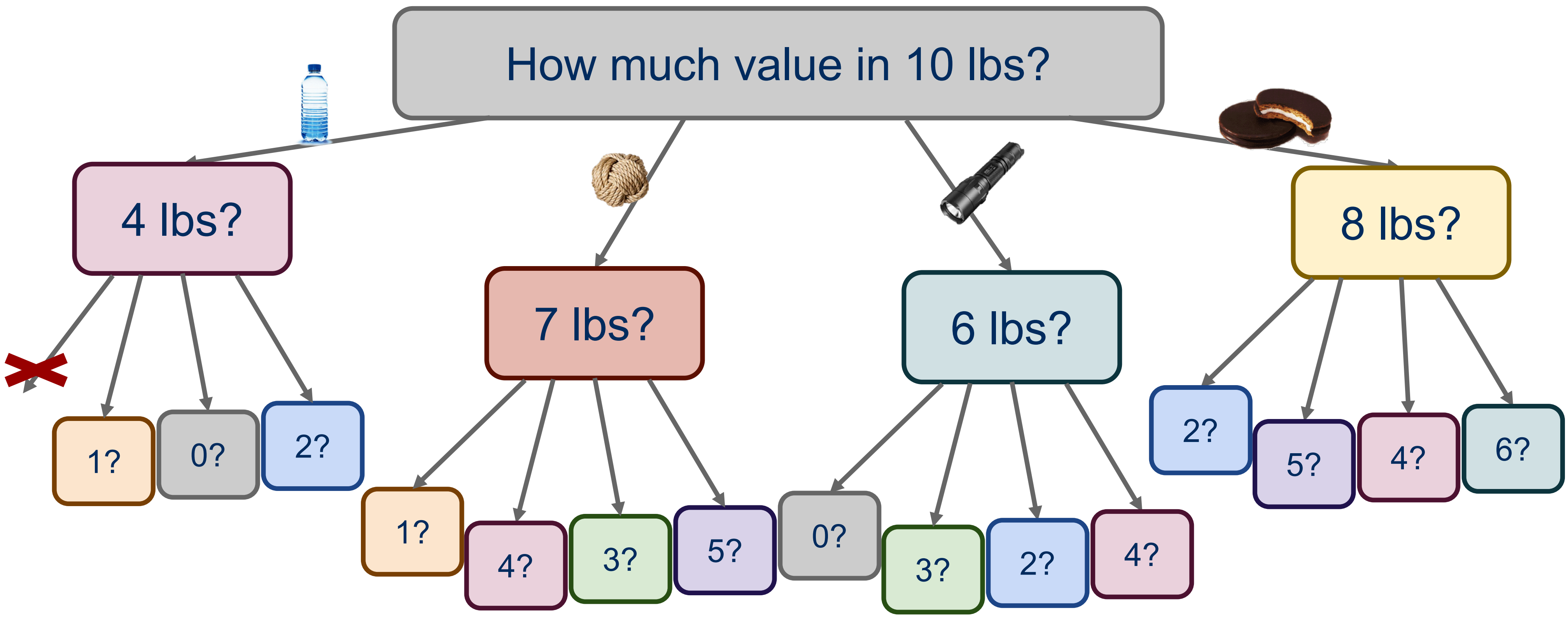
10 lb. capacity



Overlapping Subproblems!

weight:	6	3	4	2
value:	30	14	16	9

10 lb. capacity



Bottom-up Solution

				
weight:	6	3	4	2
value:	30	14	16	9

Size:	0	1	2	3	4	5	6	7	8	9	10
Max val:	0	0	9	14	18	23	30	32	39	44	48

Bottom-up solution

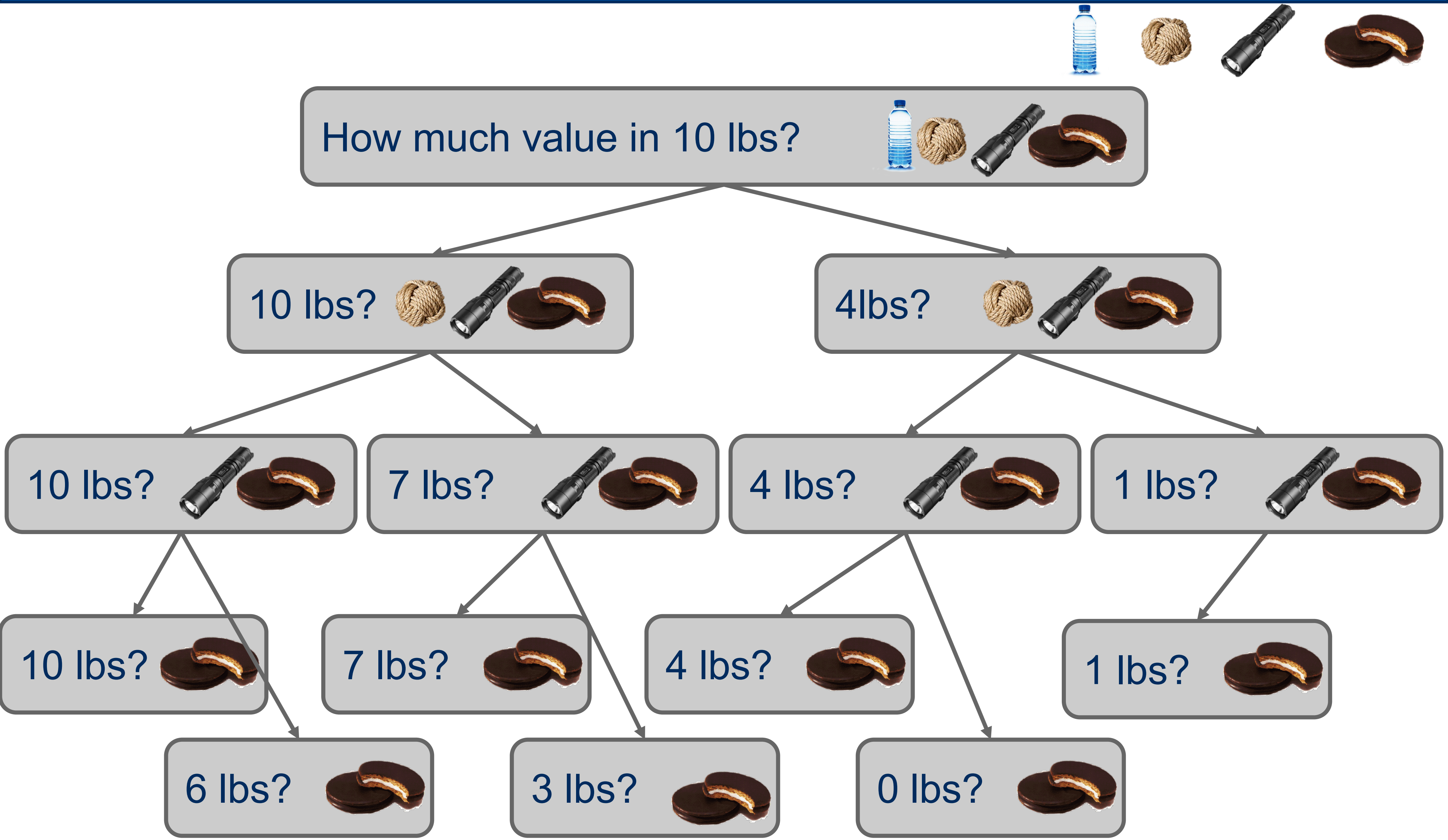
```
K[0] = 0
for (l = 1; l <= L; l++) {
      int max = 0;
      for (i = 0; i < n; i++) {
            if (wi <= l && vi + K[l - wi]) > max) {
                  max = vi + K[l - wi];
            }
      }
      K[l] = max;
}
```

The 0/1 knapsack problem

- What if we have a finite set of items that each has a weight and value?
  - Two choices for each item:
    - Goes in the knapsack
    - Is left out
- What would be our first decision?
- What suproblems emerge?

Recursive solution

weight:	6	3	4	2
value:	30	14	16	9

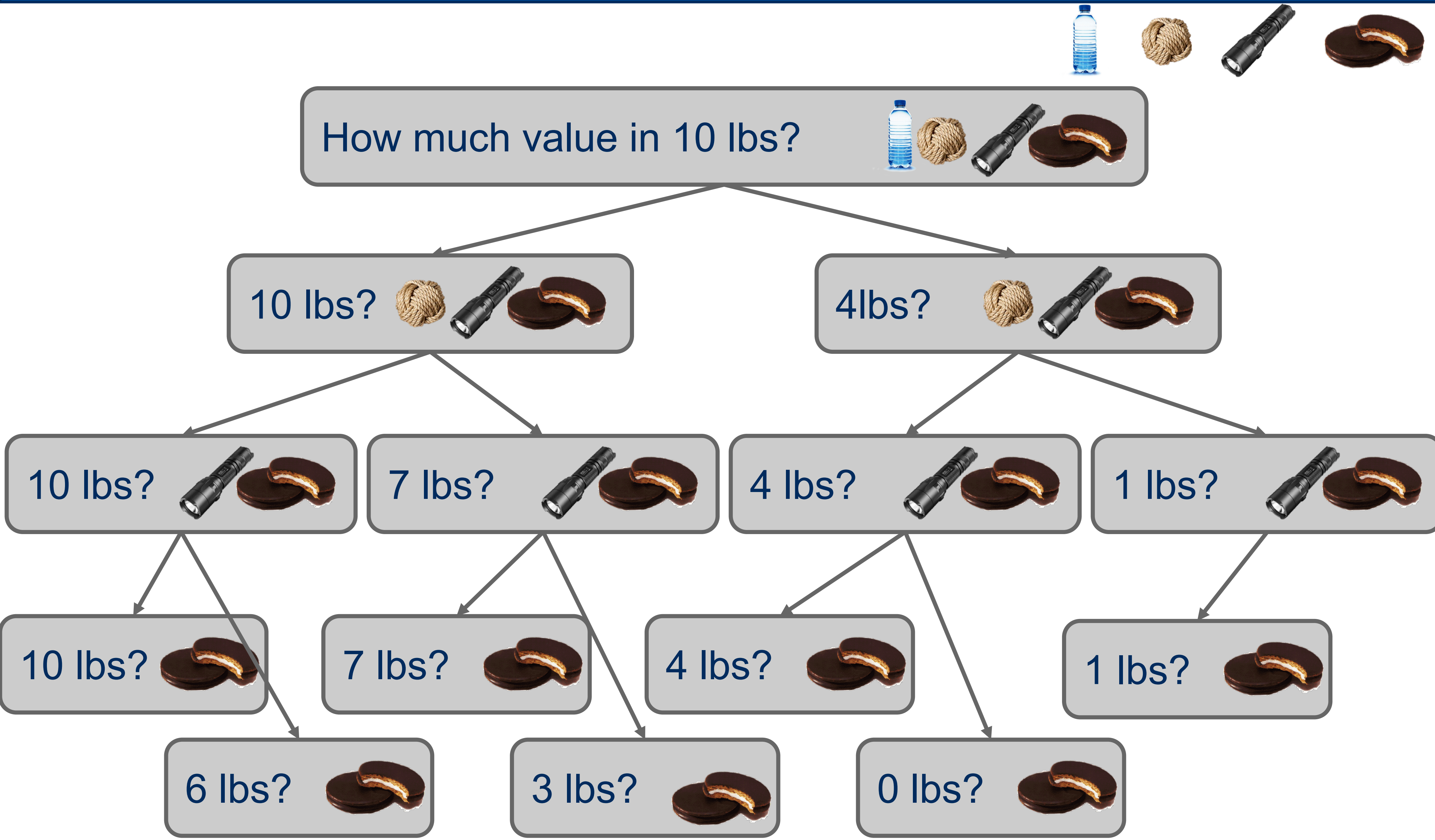


Recursive solution

```
int knapSack(int[] wt, int[] val, int L, int n) {
    if (n == 0 || L == 0) { return 0 };
    //try placing the n-1 item
    if (wt[n-1] > L) {
        return knapSack(wt, val, L, n-1)
    }
    else {
        return max( val[n-1] + knapSack(wt, val, L-wt[n-1], n-1),
                    knapSack(wt, val, L, n-1)
                   );
    }
}
```

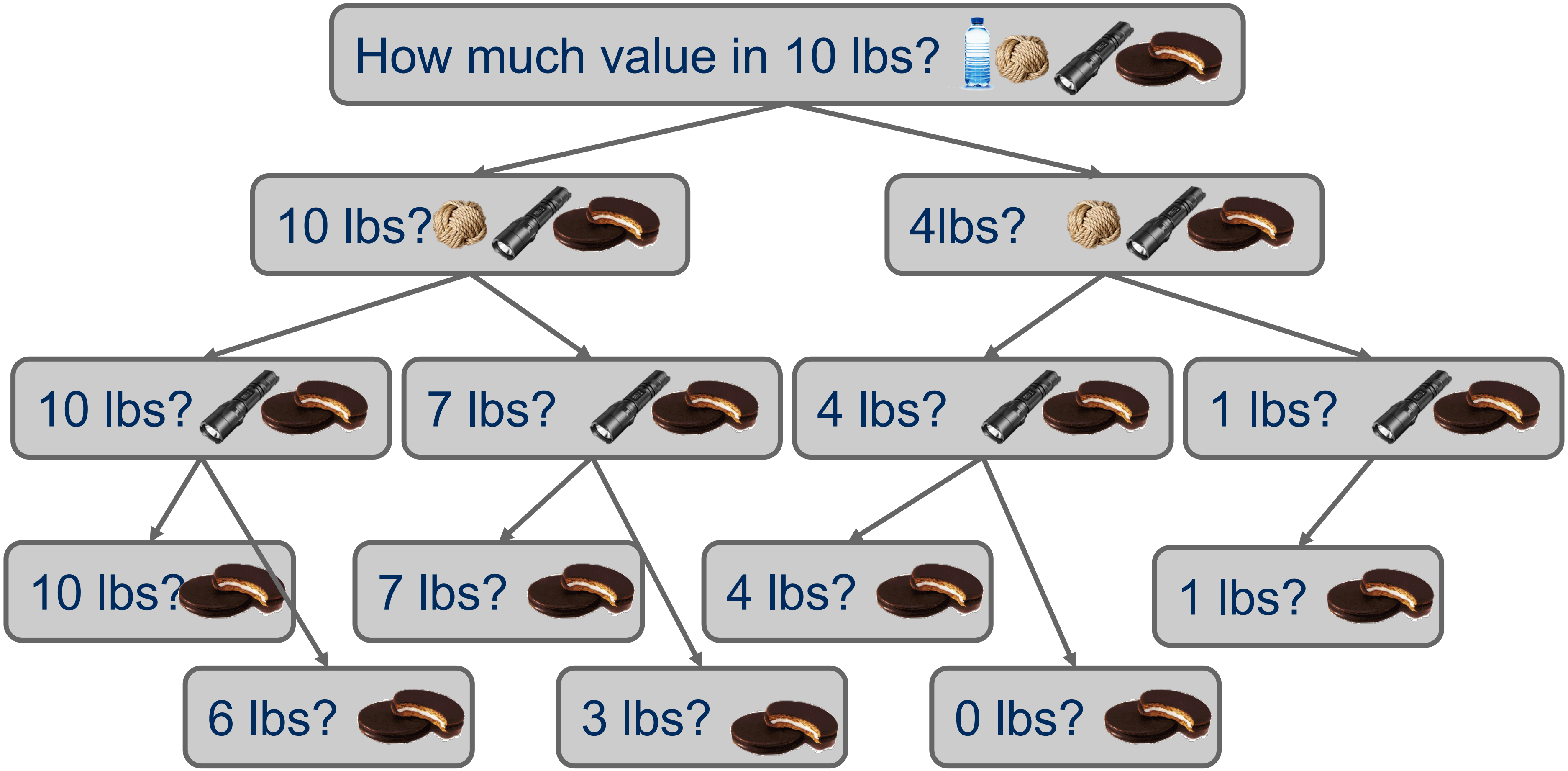
Recursive solution

weight:	6	3	4	2
value:	30	14	16	9



Subproblems

- What are the unique subproblems?



The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
K[n+1][L+1]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0											
1											
2											
3											
4											

K[i][l] is the best (max) value when only the first ***i*** items are available and only ***l*** lbs remain in the knapsack

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0										
2	0										
3	0										
4	0										

K[i][l] is the best (max) value when only the first ***i*** items are available and only ***l*** lbs remain in the knapsack

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0					
2	0										
3	0										
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0										
3	0										
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0								
3	0										
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0										
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	0	16						
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	0	16	16	30	30	30	44	46
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	0	16	16	30	30	30	44	46
4	0	0									

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	0	16	16	30	30	30	44	46
4	0	0	9	9	16	16	30	30	39	44	46

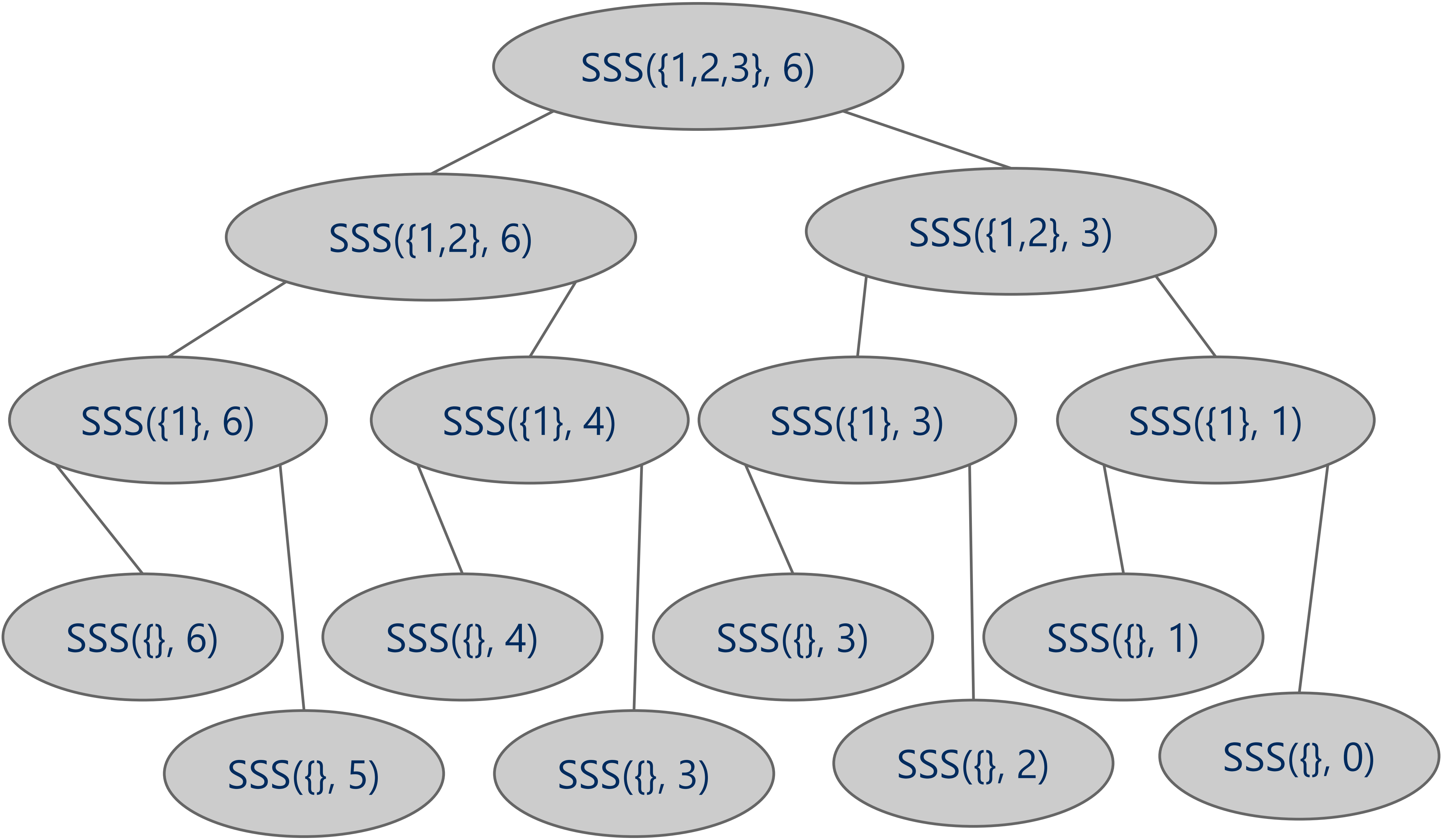
The 0/1 knapsack dynamic programming solution

```
int knapSack(int wt[], int val[], int L, int n) {
    int[][] K = new int[n+1][L+1];
    for (int i = 0; i <= n; i++) {
        for (int l = 0; l <= L; l++) {
            if (i==0 || l==0){ K[i][l] = 0 };
            //try to add item i-1
            else if (wt[i-1] > l){ K[i][l] = K[i-1][l] };
            else {
                K[i][l] = max(val[i-1] + K[i-1][l-wt[i-1]],
                                          K[i-1][l]);
            }
        }
    }
    return K[n][L];
}
```

Subset sum

- Given a set of non-negative integers S and a value k, is there a subset of S that sums to exactly k?

Subset sum calls



Subset sum recursive solution

```
boolean SSS(int set[], int sum, int n) {
    if (sum == 0)
          return true;
    if (sum != 0 && n == 0)
          return false;
    //try adding item n-1
    if (set[n-1] > sum)
          return SSS(set, sum, n-1);
    return SSS(set, sum, n-1)
          || SSS(set, sum-set[n-1], n-1);
}
```

- What would a dynamic programming table look like?

Subset sum bottom-up dynamic programming

```
boolean SSS(int set[], int sum, int n) {
    boolean[][] subset = new boolean[sum+1][n+1];

    for (int i = 0; i <= n; i++) subset[0][i] = true;
    for (int i = 1; i <= sum; i++) subset[i][0] = false;

    for (int i = 1; i <= sum; i++) {
       for (int j = 1; j <= n; j++) {
             subset[i][j] = subset[i][j-1];

             //try adding item j-1
             if (i >= set[j-1])
                    subset[i][j] ||= subset[i - set[j-1]][j-1];
       }
    }

    return subset[sum][n];
}
```

Edit Distance

- Given a string S of length n
- Given a string T of length m
- We want to find the minimum number of character changes to convert one to the other
  - called Levenshtein Distance (LD)
- Consider changes to be one of the following:
  - Change a character in a string to a different char
  - Delete a character from one string
  - Insert a character into one string

Edit Distance

- For example:
  LD("WEASEL", "SEASHELL") = 3
  - Why? Consider "WEASEL":
    - Change the W in position 1 to an S
    - Add an H in position 5
    - Add an L in position 8
  - Result is SEASHELL
    - We could also do the changes from the point of view of SEASHELL if we prefer
- How can we determine this?
  - We can define it in a recursive way initially
  - Then we will use dynamic programming to improve the run-time

Edit Distance

- We want to calculate D[n, m] where n is the length of S and m is the length of T
  - From this point of view we want to determine the distance from S to T
    - If we reverse the arguments, we get the (same) distance from T to S (but the edits may be different)

```
If n = 0        // BASE CASES
    return m (m appends will create T from S)
else if m = 0
    return n (n deletes will create T from S)
else
    Consider character n of S and character m of T
```

    - Now we have some possibilities

Edit Distance

- If characters **match**
  - return D[n-1, m-1]
    - Result is the same as the strings with the last character removed (since it matches)
  - Recursively solve the same problem with both strings one character smaller
- If characters **do not match** -- more poss. here
  - We could have a **mismatch** at that char**:**
    - return D[n-1, m-1] + 1
    - Example:
      - `S = -------------X`
      - `T = -------------Y`
    - Change X to Y, then recursively solve the same problem but with both strings one character smaller

Edit Distance

- S could have an **extra** character
  - return D[n-1, m] + 1
  - Example:
    - `S = -----------X`**`Y`**
    - `T = ------------`**`X`**
  - Delete Y, then recursively solve the same problem, with S one char smaller but with T the same size
- S could be **missing** a character there
  - return D[n, m-1] + 1
  - Example:
    - `S = ------------`**`Y`**
    - `T = -----------Y`**`X`**
  - Add X onto S, then recursively solve the same problem with S the original size and T one char smaller

Edit Distance

- Unfortunately, we don't know which of these is correct until we try them all!
- So to solve this problem we must try them all and choose the one that gives the minimum result
  - This yields 3 recursive calls for each original call (in which a mismatch occurs) and thus can give a worst-case run-time of Theta(3^n)
- How can we do this more efficiently?
  - Let's build a table of all possible values for n and m using a two-dimensional array
  - Basically we are calculating the same D[][] values but from the bottom up rather than from the top down

Edit Distance

- For each new cell D[i, j] when we have a mismatch we are taking the minimum of the cells
  - D[i-1, j] + 1
    - Append a char to S
  - D[i, j-1] + 1
    - Delete a char from S
  - D[i-1, j-1] + 1
    - Change char at this point in S if necessary
- For each new cell D[i, j] = D[i-1, j-1] if we have a match

Edit Distance

- At the end the value in the **bottom right corner** is our edit distance
- Example:
  - We are starting with PROTEIN
  - We want to generate ROTTEN
    - Note the initialization of the first row and column
    - Let's fill in the remaining squares

Edit Distance

		P	R	O	T	E	I	N
R								
O								
T								
T								
E								
N								

Edit Distance

		P	R	O	T	E	I	N
R		1	1	2	3	4	5	6
O		2	2	1	2	3	4	5
T		3	3	2	1	2	3	4
T		4	4	3	2	2	3	4
E		5	5	4	3	2	3	4
N		6	6	5	4	3	3	*3*

Edit Distance

- Why is this cool?
  - Run-time is Theta(MN)
    - As opposed to the 3^n of the recursive version
  - Unlike the pseudo-polynomial subset sum and knapsack solutions, this solution does not have any anomalous worst-case scenarios
    - There is a price, which is the space required for the matrix
    - Optimized versions can reduce this from Theta(MN) space to Theta(M+N) space

Longest Common Subsequence

- Given two sequences, return the longest common subsequence
  - A **Q** S R **J** K **V** B **I**

    Q B W F **J** **V** **I** T U
- We'll consider a relaxation of the problem and only look for the *length* of the longest common subsequence

LCS dynamic programming example

x = A Q S R J B I

y = Q B I J T U T

i\j	0	Q	B	I	J	T	U	T
0								
A								
Q								
S								
R								
J								
B								
I								

LCS dynamic programming solution

```
int LCSLength(String x, String y) {
   int[][] m = new int[x.length + 1][y.length + 1];
   for (int i=0; i <= x.length; i++) {
         for (int j=0; j <= y.length; j++) {
               if (i == 0 || j == 0) m[i][j] = 0;
               if (x.charAt(i) == y.charAt(j))
                     m[i][j] = m[i-1][j-1] + 1;
               else
                     m[i][j] = max(m[i][j-1], m[i-1][j]);
         }
      }
   return m[x.length][y.length];
}
```

Change making problem

Consider a currency with n different denominations of coins d_1, d_2, …, d_n. What is the minimum number of coins needed to make up a given value k?

So, how can we solve the change making problem optimally?

We will see a dynamic programming algorithm in the recitations